目 录

基于 C#和 OpenVINO 在英特尔独立显卡上部署 PP-TinyPose 模型

作者：杨雪锋 英特尔物联网行业创新大使

OpenVINO™ 2022.2 版开始支持英特尔独立显卡，还能通过“累计吞吐量”同时启动集成显卡 + 独立显卡助力全速 AI 推理。本文基于 C#和 OpenVINO，将 PP-TinyPose 模型部署在英特尔独立显卡上。

1.1 PP-TinyPose 模型简介

PP-TinyPose 是飞桨 PaddleDetecion 针对移动端设备优化的实时关键点检测模型，可流畅地在移动端设备上执行多人姿态估计任务。PP-TinyPose 可以基于人体 17 个关键点数据集训练后，识别人体关键点，获得人体姿态，如图 1 所示。

图 1 PP-TinyPose 识别效果图

PP-TinyPose 开源项目仓库：https://gitee.com/paddlepaddle/PaddleDetection/tree/release/2.5/configs/keypoint/tiny_pose

1.1.1 PP-TinyPose 框架

PP-TinyPose 提供了完整的人体关键点识别解决方案，主要包括行人检测以及关键点检测两部分。行人检测通过 PP-PicoDet 模型来实现，关键点识别通过 Lite-HRNet 骨干网络+DARK 关键点矫正算法来实现，如下图所示。

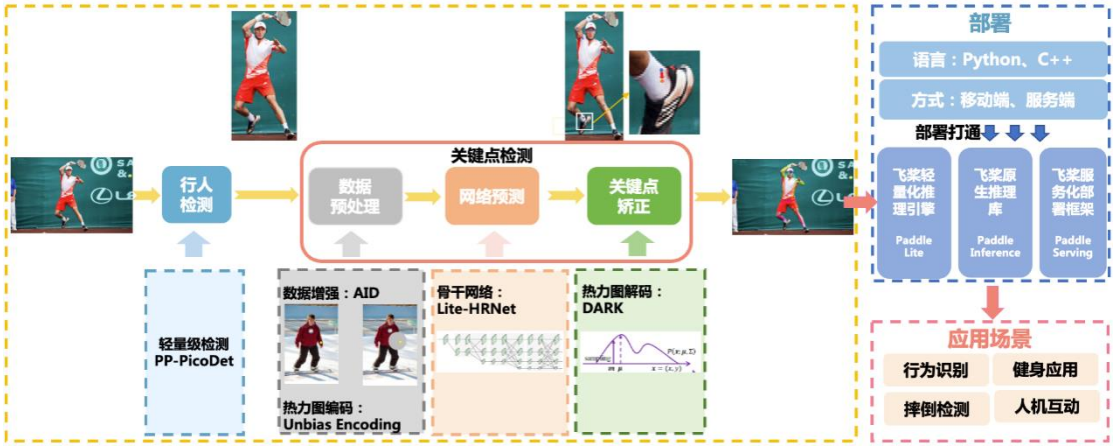

图 2 PP-TinyPose 人体关键点识别

1.2 构建开发环境

本文构建的开发环境，如下所示：

- OpenVINO™：2022.2.0

- OpenCV：4.5.5
- Visual Studio：2022
- C#框架：.NET 6.0
- OpenCvSharp：OpenCvSharp4

1.2.1 下载项目完整源代码

项目所使用的源码已在完整开源，读者可以直接克隆到本地。

```
git clone  https://gitee.com/guojin-yan/Csharp_and_OpenVINO_deploy_PP-TinyPose.git
```

1.3 在 C#中调用 OpenVINO Runtime API

由于 OpenVINO Runtime 只有 C++和 Python API 接口，需要在 C#中通过动态链接库方式调用 OpenVINO Runtime C++ API。具体教程参考《在 C#中调用 OpenVINO™ 模型》，对应的参考范例：https://github.com/guojin-yan/OpenVinoSharp.git

1.3.1 在 C#中构建 Core 类

为了更方便的使用，可以在 C#中，将调用细节封装到 Core 类中。根据模型推理的步骤，构建模型推理类：

（1）构造函数

```
public Core(string model_file, string device_name){
    // 初始化推理核心
    ptr = NativeMethods.core_init(model_file, device_name);
}
```

在该方法中，主要是调用推理核心初始化方法，初始化推理核心，读取本地模型，将模型加载到设备、创建推理请求等模型推理步骤。

（2）设置模型输入形状

```
// @brief 设置推理模型的输入节点的大小
// @param input_node_name 输入节点名
// @param input_size 输入形状大小数组
public void set_input_sharp(string input_node_name, ulong[] input_size) {
    // 获取输入数组长度
    int length = input_size.Length;
    if (length == 4) {
        // 长度为 4，判断为设置图片输入的输入参数，调用设置图片形状方法
        ptr = NativeMethods.set_input_image_sharp(ptr, input_node_name, ref input_size[0]);
    }
    else if (length == 2) {
        // 长度为 2，判断为设置普通数据输入的输入参数，调用设置普通数据形状方法
        ptr = NativeMethods.set_input_data_sharp(ptr, input_node_name, ref input_size[0]);
    }
    else {
        // 为防止输入发生异常，直接返回
        return;
```

```
            }
        }
```

（3）加载推理数据

```
        // @brief 加载推理数据
        // @param input_node_name 输入节点名
        // @param input_data 输入数据数组
        public void load_input_data(string input_node_name, float[] input_data) {
            ptr = NativeMethods.load_input_data(ptr, input_node_name, ref input_data[0]);
        }
        // @brief 加载图片推理数据
        // @param input_node_name 输入节点名
        // @param image_data 图片矩阵
        // @param image_size 图片矩阵长度
        public void load_input_data(string input_node_name, byte[] image_data, ulong image_size, int type) {
            ptr = NativeMethods.load_image_input_data(ptr, input_node_name, ref image_data[0], image_size,
type);
        }
```

加载推理数据主要包含图片数据和普通的矩阵数据，其中对于图片的预处理，也已经在 C++中进行封装，保证了图片数据在传输中的稳定性。

（5）模型推理

```
        // @brief 模型推理
        public void infer() {
            ptr = NativeMethods.core_infer(ptr);
        }
```

（6）读取推理结果数据

```
        // @brief 读取推理结果数据
        // @param output_node_name 输出节点名
        // @param data_size 输出数据长度
        // @return 推理结果数组
        public T[] read_infer_result<T>(string output_node_name, int data_size) {
            // 获取设定类型
            string t = typeof(T).ToString();
            // 新建返回值数组
            T[] result = new T[data_size];
            if (t == "System.Int32") { // 读取数据类型为整形数据
                int[] inference_result = new int[data_size];
                NativeMethods.read_infer_result_I32(ptr, output_node_name, data_size, ref
inference_result[0]);
                result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
                return result;
            }
            else { // 读取数据类型为浮点型数据
                float[] inference_result = new float[data_size];
```

```
                NativeMethods.read_infer_result_F32(ptr, output_node_name, data_size, ref
inference_result[0]);
                result = (T[])Convert.ChangeType(inference_result, typeof(T[]));
                return result;
            }
        }
```

在读取模型推理结果时，支持读取整形数据和浮点型数据。

（7）清除地址

```
        // @brief 删除创建的地址
        public void delet() {
            NativeMethods.core_delet(ptr);
        }
```

完成上述封装后，在 C#平台下，调用 Core 类，就可以方便实现 OpenVINO 推理程序了。

1.4 下载并转换 PP-PicoDet 模型

1.4.1 PP-PicoDet 模型简介

Picodet_s_320_lcnet_pedestrian Paddle 格式模型信息如下表所示，其默认的输入为动态形状，需要将该模型的输入形状变为静态形状。

表 1 Picodet_s_320_lcnet_pedestrian Paddle 格式模型信息

	Input	Output	
名称	x	concat_8.tmp_0	transpose_8.tmp_0
形状	[bath_size, 3, 320, 320]	[bath_size, 2125, 4]	[bath_size, 1, 2125]
数据类型	Float32	Float32	Float32

1.4.2 模型下载与转换

第一步：下载模型：

命令行直接输入以下模型导出代码，使用 PaddleDetecion 自带的方法，下载预训练模型并将模型转为导出格式。

导出 picodet_s_320_lcnet_pedestrian 模型：

```
python tools/export_model.py -c
configs/picodet/application/pedestrian_detection/picodet_s_320_lcnet_pedestrian.yml -o export.benchmark=False
export.nms=False
weights=https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/picodet_s_320_lcnet_pedestrian.p
dparams --output_dir=output_inference
```

导出 picodet_s_192_lcnet_pedestrian 模型：

```
python tools/export_model.py -c
configs/picodet/application/pedestrian_detection/picodet_s_192_lcnet_pedestrian.yml -o export.benchmark=False
export.nms=False
```

```
weights=https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/picodet_s_192_lcnet_pedestrian.pdparams --output_dir=output_inference
```

此处导出模型的命令与我们常用的命令导出增加了 export.benchmark=False 和 export.nms=False 两个指令，主要是关闭模型后处理以及打开模型极大值抑制。如果不关闭模型后处理，模型会增加一个输入，且在模型部署时会出错。

第二步，将模型转换为 ONNX 格式：

该方式需要安装 paddle2onnx 和 onnxruntime 模块。导出方式比较简单，比较注意的是需要指定模型的输入形状，用于固定模型批次的大小。在命令行中输入以下指令进行转换：

```
paddle2onnx --model_dir output_inference/picodet_s_320_lcnet_pedestrian --model_filename model.pdmodel --params_filename model.pdiparams --input_shape_dict "{'image':[1,3,320,320]}" --opset_version 11 --save_file picodet_s_320_lcnet_pedestrian.onnx
```

第三步：转换为 IR 格式

利用 OpenVINO™ 模型优化器，可以实现将 ONNX 模型转为 IR 格式

```
mo --input_model picodet_s_320_lcnet_pedestrian.onnx --input_shape [1,3,256,192] --data_type FP16
```

1.5 下载并转换 PP-TinyPose 模型

1.5.1 PP-TinyPose 模型简介

PP-TinyPose 模型信息如下表所示，其默认的输入为动态形状，需要将该模型的输入形状变为静态形状。

表 2 PP-TinyPose 256×192 Paddle 模型信息

	Input	Output	
名称	image	conv2d_441.tmp_1	argmax_0.tmp_0
形状	[bath_size, 3, 256, 192]	[bath_size, 17, 64, 48]	[bath_size,17]
数据类型	Float32	Float32	Int64

1.5.2 模型下载与转换

第一步：下载模型：

命令行直接输入以下代码，或者浏览器输入后面的网址即可。

```
wget https://bj.bcebos.com/v1/paddledet/models/keypoint/tinypose_enhance/tinypose_256x192.zip
```

下载好后将其解压到文件夹中，便可以获得 Paddle 格式的推理模型。

第二步：转换为 ONNX 格式：

该方式需要安装 paddle2onnx 和 onnxruntime 模块。在命令行中输入以下指令进行转换，其中转换时需要指定 input_shape，否者推理时间会很长：

```
paddle2onnx --model_dir output_inference/tinypose_256_192/paddle --model_filename model.pdmodel --params_filename model.pdiparams --input_shape_dict "{'image':[1,3,256,192]}" --opset_version 11 --save_file tinypose_256_192.onnx
```

第三步：转换为 IR 格式

利用 OpenVINO™ 模型优化器，可以实现将 ONNX 模型转为 IR 格式。

```
cd .\openvino\tools
mo --input_model paddle/model.pdmodel --input_shape [1,3,256,192] --data_type FP16
```

1.6 编写 OpenVINO 推理程序

1.6.1 实现行人检测

第一步：初始化 PicoDet 行人识别类

```
// 行人检测模型
string mode_path_det =
@"E:\Text_Model\TinyPose\picodet_v2_s_320_pedestrian\picodet_s_320_lcnet_pedestrian.onnx";
// 设备名称
string device_name = "CPU";
PicoDet pico_det = new PicoDet(mode_path_det, device_name);
```

首先初始化行人识别类，将本地模型读取到内存中，并将模型加载到指定设备中。

第二步：设置输入输出形状

```
Size size_det = new Size(320, 320);
pico_det.set_shape(size_det, 2125);
```

根据我们使用的模型，设置模型的输入输出形状。

第三步：实现行人检测

```
// 测试图片
string image_path = @"E:\Git_space\基于 Csharp 和 OpenVINO 部署 PP-TinyPose\image\demo_3.jpg";
Mat image = Cv2.ImRead(image_path);
List<Rect> result_rect = pico_det.predict(image);
```

在进行模型推理时，使用 OpenCvSharp 读取图像，然后带入预测，最终获取行人预测框。最后将行人预测框绘制到图片上，如下图所示。

图 3 行人位置预测结果

1.6.2 实现人体姿态识别

第一步：初始化 P 人体姿势识别 PPTinyPose 类

```
// 关键点检测模型
// onnx 格式
string mode_path_pose = @"E:\Text_Model\TinyPose\tinypose_128_96\tinypose_128_96.onnx";
// 设备名称
string device_name = "CPU";
PPTinyPose tiny_pose = new PPTinyPose(mode_path_pose, device_name);
```

首先初始化人体姿势识别 PPTinyPose 类，将本地模型读取到内存中，并加载到设备上。

第二步：设置输入输出形状

```
Size size_pose = new Size(128, 96);
tiny_pose.set_shape(size_pose);
```

PP-TinyPose 模型输入与输出有对应关系，因此只需要设置输入尺寸

第三步：实现姿势预测

```
// 测试图片
string image_path = @"E:\Git_space\基于 Csharp 和 OpenVINO 部署 PP-TinyPose\image\demo_3.jpg";
Mat image = Cv2.ImRead(image_path);
Mat result_image = tiny_pose.predict(image);
```

在进行模型推理时，使用 OpenCvSharp 读取图像，然后带入预测，最终获取人体姿势结果，如下图所示。

图 4 人体姿态绘制效果图

1.6.3 推理速度测试

本项目在蝰蛇峡谷上完成测试，CPU 为 i7-12700H，自带锐炬®集成显卡；独立显卡为

英特尔®锐炫® A770M 独立显卡+16G 显存，如下图所示。

图 5 螳螂峡谷

测试代码已开源：https://gitee.com/guojin-yan/Csharp_and_OpenVINO_deploy_PP-TinyPose.git

测试结果如下表所示

表 3 PP-PicoDet 与 PP-TinyPose 模型运行时间(ms)

推理设备	模型名称	PP-PicoDet 320×320				PP-TinyPose 256×192				FPS
	模型格式	模型读取	加载数据	模型推理	结果处理	模型读取	加载数据	模型推理	结果处理	
i7-12700H	IR-FP16	159.74	1.10	2.97	0.08	322.23	0.84	5.12	1.67	85
A770M	IR-FP16	5250.30	1.36	3.77	0.01	12575.64	1.01	8.95	1.57	60

注： 模型读取：读取本地模型，加载到设备，创建推理通道；

加载数据：将待推理数据进行处理并加载到模型输入节点；

模型推理：模型执行推理运算；

结果处理：在模型输出节点读取输出数据，并转化为我们所需要的结果数据。

1.7 总结与未来工作展望

本文完整介绍了在 C#中基于 OpenVINO 部署 PP-TinyPose 模型的完整流程，并开源了完整的项目代码。

从表 3 的测试结果可以看到，面对级联的小模型，由于存在数据从 CPU 传到 GPU，GPU 处理完毕后，结果从 GPU 传回 CPU 的时间消耗，独立显卡相对 CPU 并不具备明显优势。

未来改进方向：

1. 借助 OpenVINO 预处理 API，将预处理和后处理集成到 GPU 中去。
2. 借助 OpenVINO 异步推理 API，提升 GPU 利用率
3. 仔细分析 CPU 和 GPU 之间的数据传输性能瓶颈，尝试锁页内存、异步传输等优化

技术，“隐藏”CPU 和 GPU 之间的数据传输时间消耗。